# *School of Electronic Engineering*

| | |
|---|---|
| Student Name | Sakthignana Sundaram Somaskandan |
| Student Number | 14346091 |
| Programme | Bachelor of Engineering: Electronics and Computers Engineering |
| Project Title | Algorithms for Engineers Project Report |
| Module code | EE324 |
| Lecturer | Dr. Conor McArdle |
| Project Due Date | 9/12/2016 |

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying is a grave and serious offence in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references.

I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified.

This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. I have read and understood the referencing guidelines found at http://www.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.

I understand that I may be required to discuss with the module lecturer/s the contents of this submission.

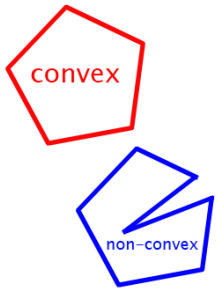
I/me/my incorporates we/us/our in the case of group work, which is signed by all of us.

Signed: Sakthignana Sundaram Somaskandan

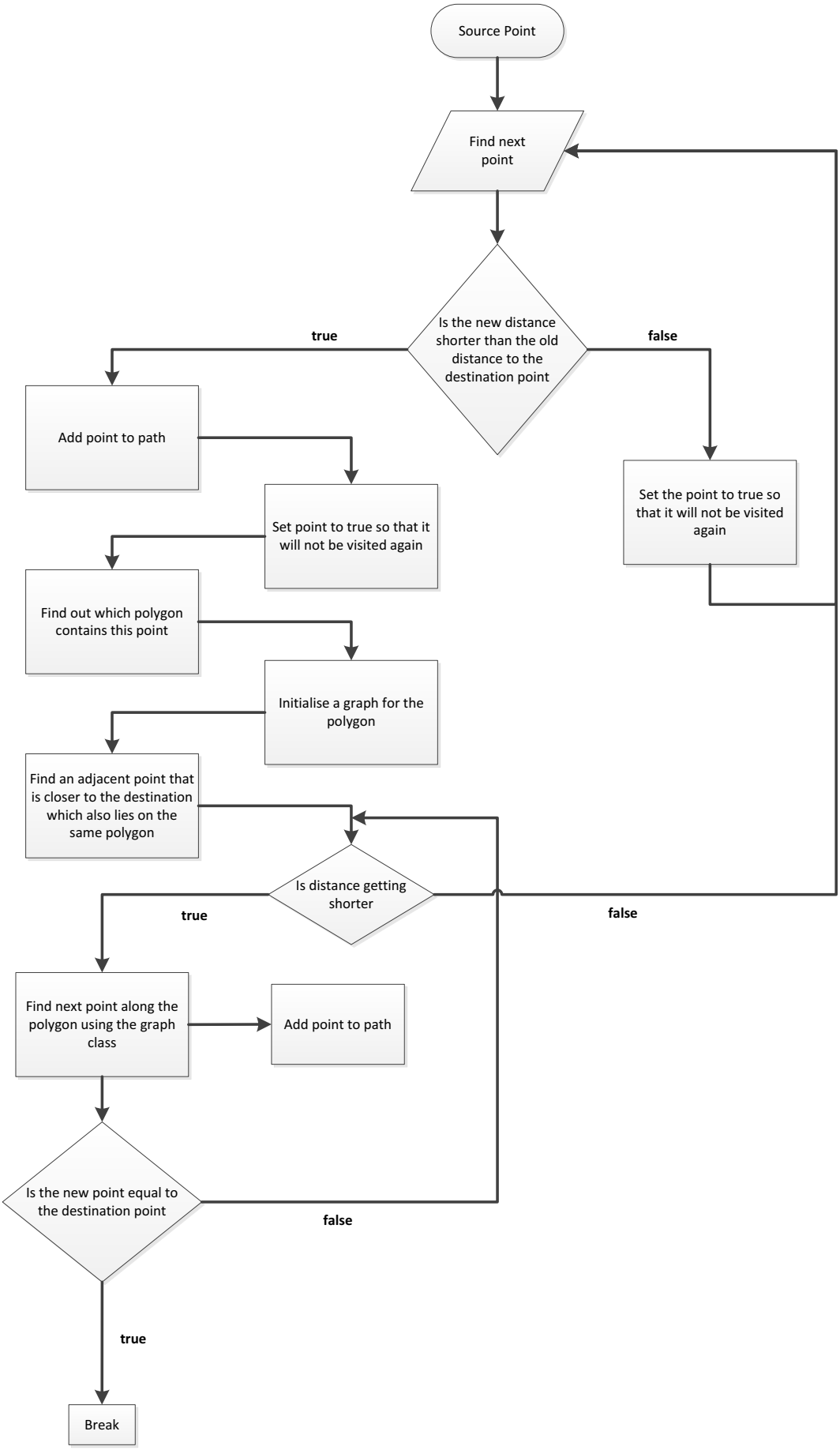
## Table of Contents

# *Description of Algorithm*

The algorithm that I used to find a path in any given map relates closely to the nearest neighbour algorithm from the travelling salesman problem (TSP) but with an addition of another class called the Graph. The graph class basically sets the vertices and the edges of a given polygon (convex hull) and returns the adjacent vertices to a given vertex. The path obtained would not necessarily be the optimal or the shortest path but it's a decent valid path. It can almost be considered as an extension to the nearest neighbour algorithm with a more focused approach. This algorithm requires the convex hull of each polygon in a map to be acquired before actually proceeding to find a path. This is because a polygon could have sharp edges but the convex hull of the same polygon wouldn't contain any sharp edges or dents as depicted in the figure below:




The basic concept of my algorithm is for a given source point, using the nearest neighbour algorithm the closest point to the source point is calculated. Once this point has been obtained, a number of checks is done on the point before the point is added to the list of points which eventually will construct a path. There is only two checks involved for this algorithm and one of the checks is to test if the distance (distance between the current point and the destination point) is getting shorter and shorter each time so that the path is getting closer and closer to the destination point and another check is to test if the current point is equal to the destination point, this is to break out of the main loop. The block diagram for this algorithms is shown below:


Source Point
Find next point
Is the new distance shorter than the old distance to the destination point
true
false
Add point to path
Set point to true so that it will not be visited again
Set the point to true so that it will not be visited again
Find out which polygon contains this point
Initialise a graph for the polygon
Find an adjacent point that is closer to the destination which also lies on the same polygon
Is distance getting shorter
true
false
Find next point along the polygon using the graph class
Add point to path
Is the new point equal to the destination point
false
true
Break

The classes used by this algorithm (*'FindPath.java'* in the appendix) are as follows:

1. *Graph.java*
2. *TSP_NN.java*
3. *DistanceMatrix.java*

This algorithm requires and calls numerous sub routines inside and outside of the class.

Inside the class *'FindPath.java':*

Prior to initialisation of path finding algorithm, all the points must be loaded in correctly and ready to be used and this is done by the method *'insert'* in the class *'FindPath.java'* (see appendix for code). The convex hull points are put in two arraylists, each arraylist holds the points in a slightly different way. One arraylist holds all the convex hull points in the map and the other holds arrays of Point2D, where each array refers to a different convex hull polygon in the map. The source and the destination points are also loaded in using different methods called *'setSourcepoint'* and *'setDestinationpoint'*. The next method in the *'FindPath.java'* is really important as this computes the index of the polygon which has the given point. It does this by checking if the point exists in the arraylist of arrays and returns the index of the array in the arraylist, in which the given point exists. Another method called *'setDistance'* is used to calculate the total distance of the path. It executes this by using the *'Math'* class *'sqrt'* function that is defined in the *'Point2D'* class. And the last two methods in this class are the *'draw'* methods. One is used to draw the path without connecting the source and the destination points and the other is to draw individual points on the canvas.

Outside the class *'FindPath.java'*:

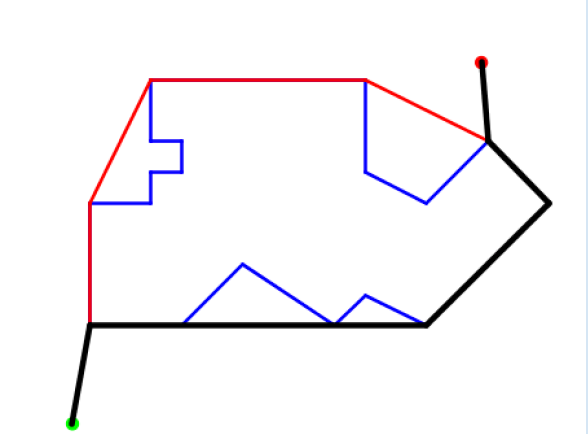
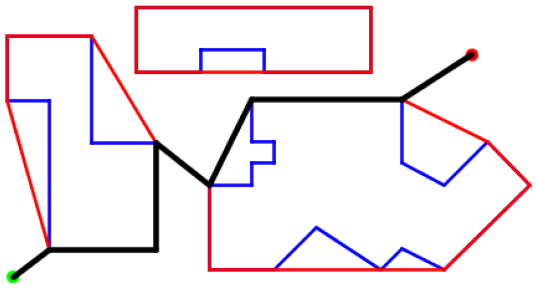
The Graph class is basically used to identify the adjacent vertices to a given vertex. This is achieved by making a 2 dimensional array and setting an element inside the 2D array *'true'* when an edge exists between two vertices. The edges were added using the method *'addEdge'* in the graph class, it receives two parameters and it serves as the two vertices at either end of the edge. To retrieve the adjacent vertices to a vertex, the method *'adj'* is used. it takes in a vertex as a parameter and returns an arraylist of integers containing the index of the adjacent vertices. It accomplishes this by checking if the given vertex (input) and another vertex in the polygon is set to *'true'*.

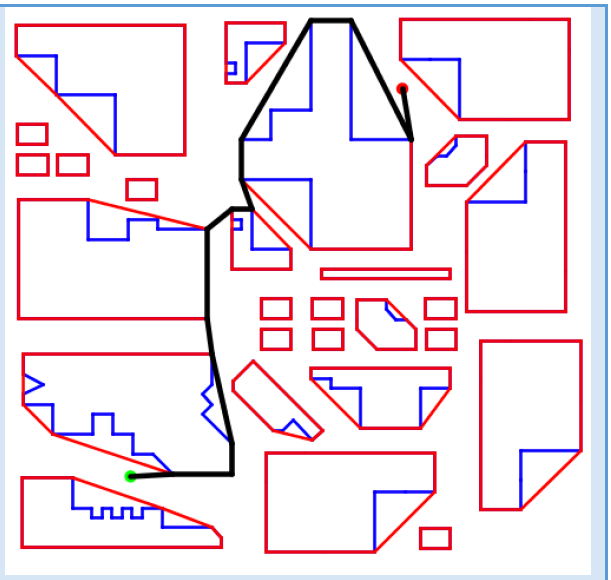
The TSP_NN class is used to get the closest point to a given point. The TSP_NN constructor loads all the points into a private variable so that it can be used locally inside the TSP_NN class. The nearest neighbour works by storing the distance between two points in a 2 dimensional array and uses the distances stored in that array to get the shortest distance relative to the current point and in turn gets the closest point. The closest point is calculated by the *'findNextpoint'* method in the TSP_NN class. There is also another method called *'setTrue'* that is useful when a point needs to be set *'true'* from outside the TSP_NN class so that it will not be visited twice. This method is used when the path finding algorithm finds a point that results in somewhat a longer distance, that point needs to be set *'true'* from outside the class.

The DistanceMatrix class initialises a 2 dimensional array and calculates the distance between a point and all the other points in the map and stores it in the 2 dimensional array

for ease of retrieval and usage. The DistanceMatrix class constructor is used to initialise the size of the 2 dimensional array. The method *'addPoint'* in the DistanceMatrix class is used to add in a point and compute and store the distance between that point and all the other points that exist in the array, using the distance formula that is defined in the Point2D class. To retrieve the distance between any two valid points, the *'getDistance'* method can be used. This method takes in 2 variables (two points) and returns the distance between those 2 parameters.

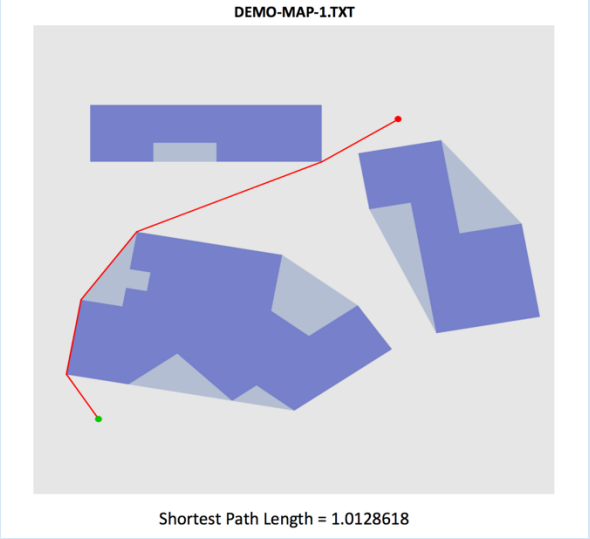
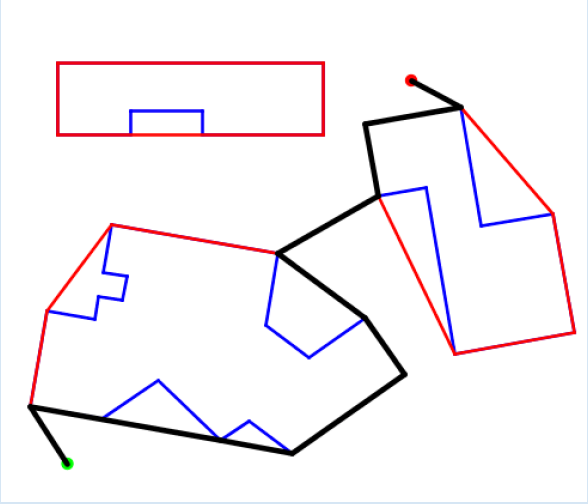
As any other algorithm, this algorithm has strengths and weaknesses: the strengths are the complexity and the efficiency of the algorithm, this algorithm is not computationally expensive since the use of java in-built classes have been minimised, e.g. Math class; a weakness is that this algorithm is not perfect and there are situations where this algorithm will fail and this will be discussed in detail later in the report. However, this algorithm works 90% - 95% of the times. The demonstration of this algorithm is shown below using the test maps that was provided:

| *Test Map Number* | *My Algorithm Solution* |
|---|---|
| *Test Map 0* |  <br>$Execution\ time\ (ms)$: 19<br>$Length\ of\ path$: 1.19815166186768 |
| *Test Map 1* |  <br>$Execution\ time\ (ms)$: 39<br>$Length\ of\ path$: 1.0536514793028522 |

| *Test Map 2* |  |
|---|---|

*Execution time (ms)*: 316

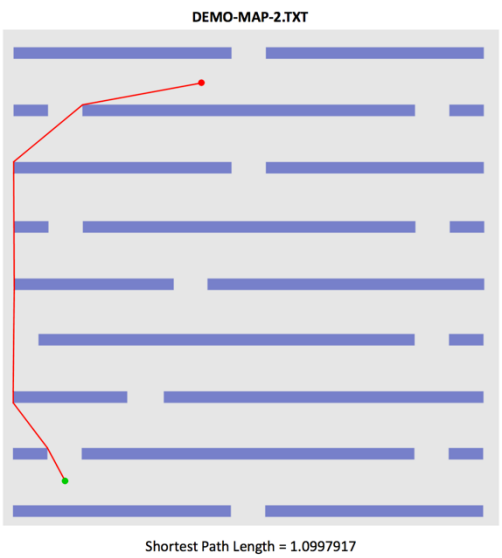
*Length of path*: 1.4160792030026226

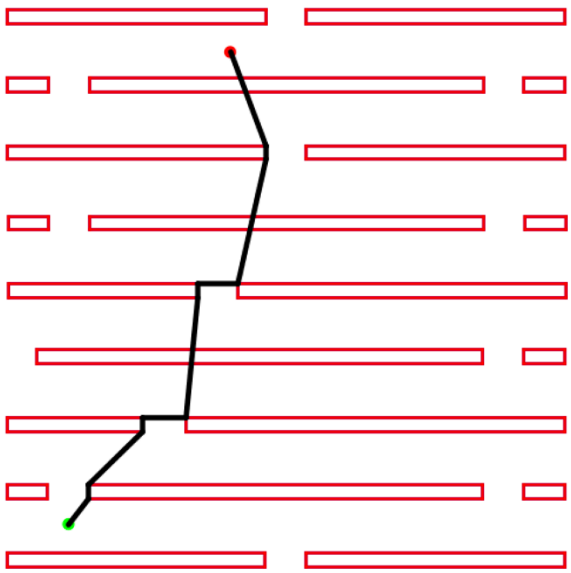
| *Optimal Solution* | *My Algorithm Solution* |
|---|---|
|   |  |

Shortest Path Length = 1.0128618

*Execution time (ms)*: 38

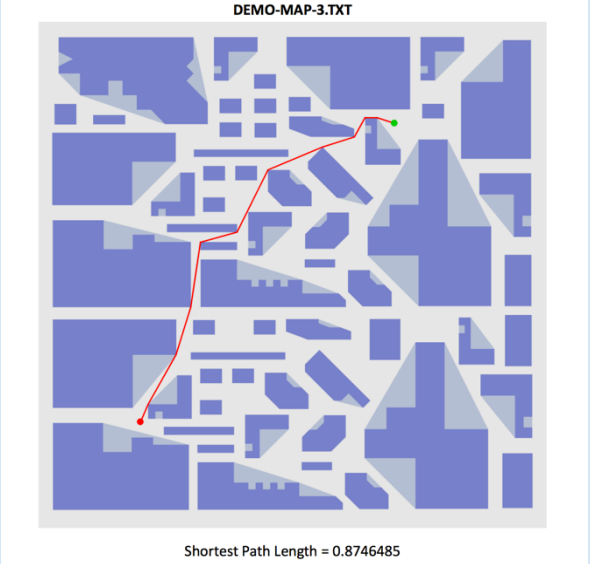
*Length of path*: 1.6518031903513315

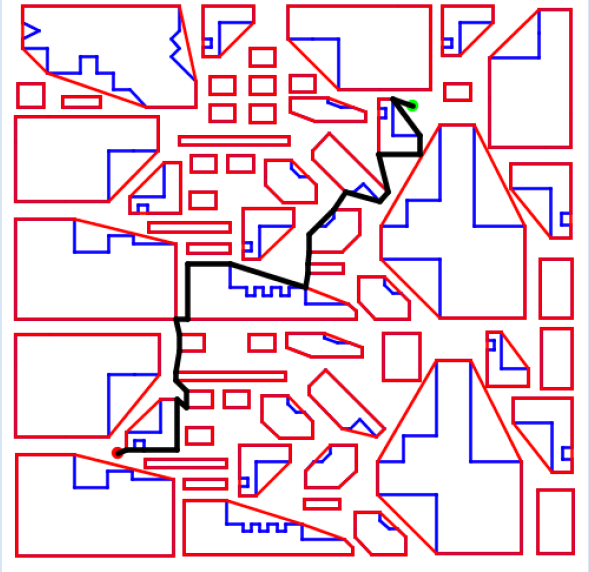


*Execution time (ms)*: 179

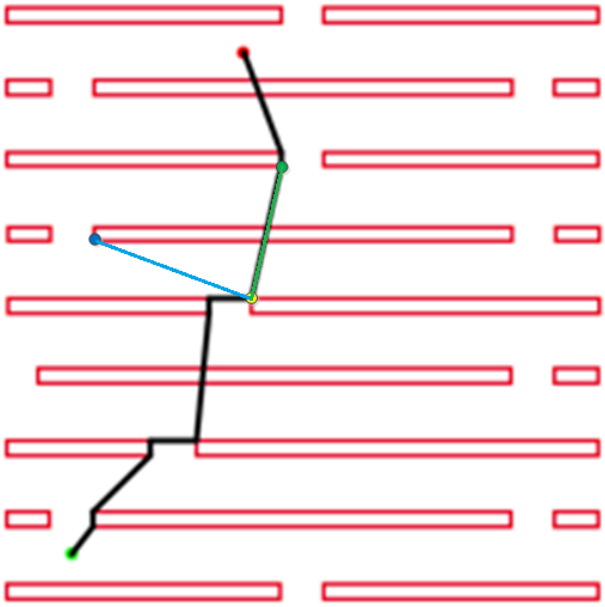
*Length of path*: 1.0123272423350442



*Execution time (ms)*: 696

*Length of path*: 1.235976257634803



The algorithm fails for DEMO-MAP-2 because the algorithm uses the nearest neighbour and since the vertices of the convex hull of the polygons in the map are far apart from each other, when finding the closest point to the current point the algorithm doesn't know there is a polygon in between. This occurs because the distance between two points is used to compute the closest point to the current point. As you can see in the figure above, the distance between the yellow point and the blue point (blue line) is greater than the distance between the yellow point and the green point (green line) and therefore picks the point that has a shorter distance without realising that the path is cutting through a hull.

The length of the path found for the DEMO maps are not even close to the optimal solution. The difference in percentage is derived using the following formula:

$$Difference\ (\%) = \frac{my\ solution - optimal\ solution}{optimal\ solution} \times 100$$

$DEMO - MAP - 1$:

$$Difference\ (\%) = \frac{1.6518031903513315 - 1.0128618}{1.0128618} \times 100 = 63.08\ \%$$

$DEMO - MAP - 2$:

$$Difference\ (\%) = \frac{1.0123272423350442 - 1.0997917}{1.0997917} \times 100 = -7.9528\ \%$$

This value is invalid because the path found is invalid as you can see above in the table (DEMO-MAP-2). The path cuts through a hull multiple times.

$DEMO - MAP - 3$:

$$Difference\ (\%) = \frac{1.235976257634803 - 0.8746485}{0.8746485} \times 100 = 41.3\ \%$$

# *Implementation Design*

The classes used are as follows:

1. **DistanceMatrix.java**

This class is used to compute the distance between a point and all the other points in the map and store it in a 2 dimensional array for use later in different class.

2. **FindPath.java**

This class is used to determine a path between the given source and destination points without cutting through any convex hulls and also at the same time try and get the shortest route possible for the algorithm.

3. **GrahamScan.java**

This class is used to get the convex hull of a given polygon using the grahamscan method.

4. **Graph.java**

This class is used to initialise and make a digital version of a given polygon. This is done so the program is aware of the shape of the polygon, e.g. what vertices are connected to a given vertex in a polygon.

5. **MapFileReader.java**

This class is used to read data from an external file and store the data inside the program to make it available for the computations done in the program

6. **Point2D.java**

This class is used to store the x and y co-ordinates of a point and do some calculations on the given point, i.e. cross product calculation for turning direction determination.

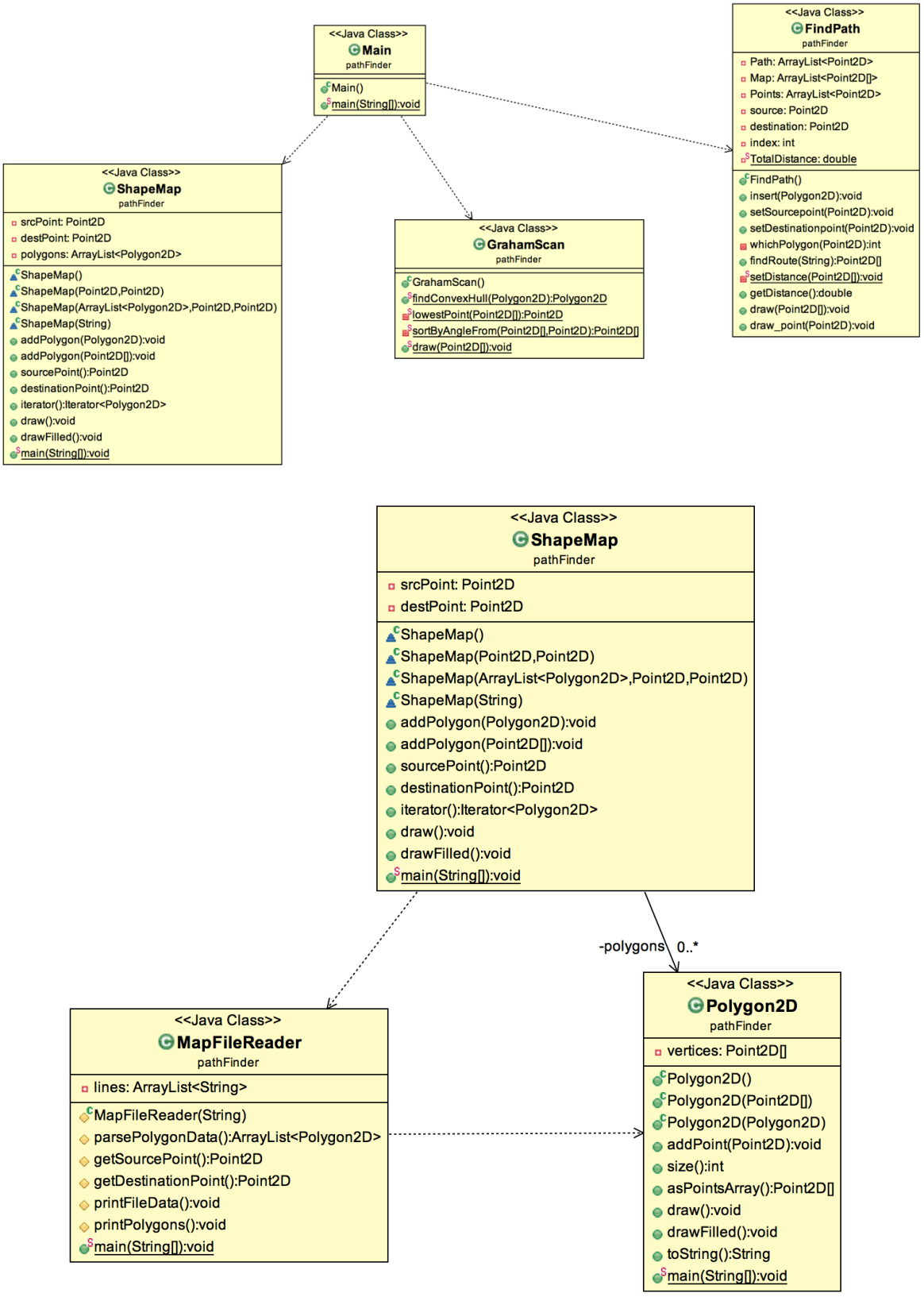
7. **Polygon2D.java**

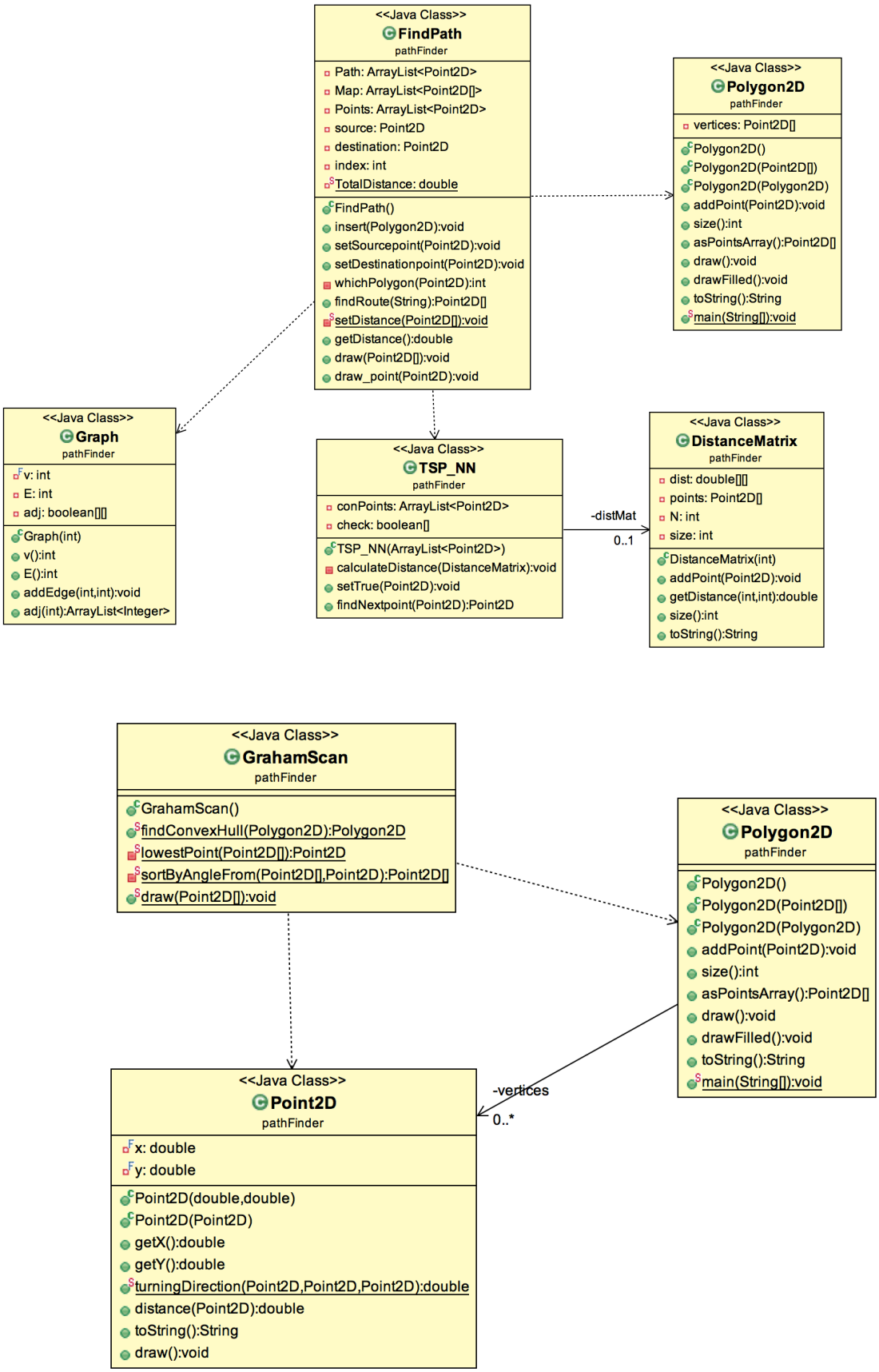
This class stores the vertices of a polygon in an array of Point2D object.

8. **TSP_NN.java**

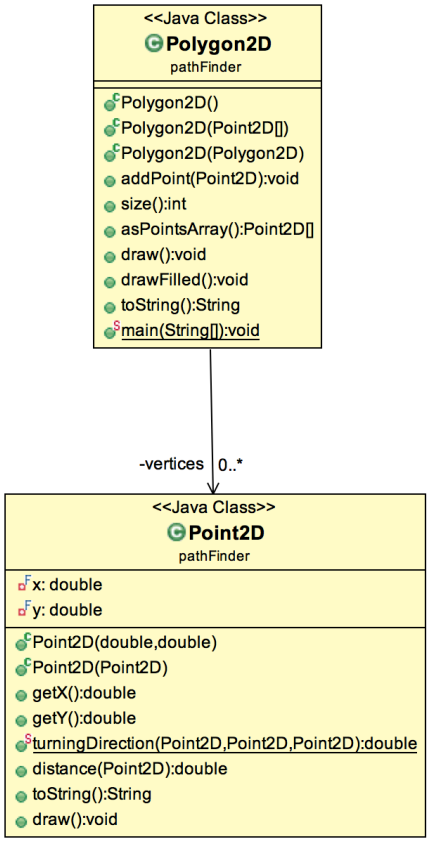
This class is used to determine the closest unvisited point to a given point.

The relationship between the classes are outlined below:


<<Java Class>>
Main
pathFinder
Main()
main(String[]):void
<<Java Class>>
FindPath
pathFinder
Path: ArrayList<Point2D>
Map: ArrayList<Point2D[]>
Points: ArrayList<Point2D>
source: Point2D
destination: Point2D
index: int
TotalDistance: double
FindPath()
insert(Polygon2D):void
setSourcepoint(Point2D):void
setDestinationpoint(Point2D):void
whichPolygon(Point2D):int
findRoute(String):Point2D[]
setDistance(Point2D[]):void
getDistance():double
draw(Point2D[]):void
draw_point(Point2D):void
<<Java Class>>
ShapeMap
pathFinder
srcPoint: Point2D
destPoint: Point2D
polygons: ArrayList<Polygon2D>
ShapeMap()
ShapeMap(Point2D,Point2D)
ShapeMap(ArrayList<Polygon2D>,Point2D,Point2D)
ShapeMap(String)
addPolygon(Polygon2D):void
addPolygon(Point2D[]):void
sourcePoint():Point2D
destinationPoint():Point2D
iterator():Iterator<Polygon2D>
draw():void
drawFilled():void
main(String[]):void
<<Java Class>>
GrahamScan
pathFinder
GrahamScan()
findConvexHull(Polygon2D):Polygon2D
lowestPoint(Point2D[]):Point2D
sortByAngleFrom(Point2D[],Point2D):Point2D[]
draw(Point2D[]):void
<<Java Class>>
ShapeMap
pathFinder
srcPoint: Point2D
destPoint: Point2D
ShapeMap()
ShapeMap(Point2D,Point2D)
ShapeMap(ArrayList<Polygon2D>,Point2D,Point2D)
ShapeMap(String)
addPolygon(Polygon2D):void
addPolygon(Point2D[]):void
sourcePoint():Point2D
destinationPoint():Point2D
iterator():Iterator<Polygon2D>
draw():void
drawFilled():void
main(String[]):void
-polygons
0..*
<<Java Class>>
MapFileReader
pathFinder
lines: ArrayList<String>
MapFileReader(String)
parsePolygonData():ArrayList<Polygon2D>
getSourcePoint():Point2D
getDestinationPoint():Point2D
printFileData():void
printPolygons():void
main(String[]):void
<<Java Class>>
Polygon2D
pathFinder
vertices: Point2D[]
Polygon2D()
Polygon2D(Point2D[])
Polygon2D(Polygon2D)
addPoint(Point2D):void
size():int
asPointsArray():Point2D[]
draw():void
drawFilled():void
toString():String
main(String[]):void


<<Java Class>>
FindPath
pathFinder
Path: ArrayList<Point2D>
Map: ArrayList<Point2D[]>
Points: ArrayList<Point2D>
source: Point2D
destination: Point2D
index: int
TotalDistance: double
FindPath()
insert(Polygon2D):void
setSourcepoint(Point2D):void
setDestinationpoint(Point2D):void
whichPolygon(Point2D):int
findRoute(String):Point2D[]
setDistance(Point2D[]):void
getDistance():double
draw(Point2D[]):void
draw_point(Point2D):void
<<Java Class>>
Polygon2D
pathFinder
vertices: Point2D[]
Polygon2D()
Polygon2D(Point2D[])
Polygon2D(Polygon2D)
addPoint(Point2D):void
size():int
asPointsArray():Point2D[]
draw():void
drawFilled():void
toString():String
main(String[]):void
<<Java Class>>
Graph
pathFinder
v: int
E: int
adj: boolean[][]
Graph(int)
v():int
E():int
addEdge(int,int):void
adj(int):ArrayList<Integer>
<<Java Class>>
TSP_NN
pathFinder
conPoints: ArrayList<Point2D>
check: boolean[]
TSP_NN(ArrayList<Point2D>)
calculateDistance(DistanceMatrix):void
setTrue(Point2D):void
findNextpoint(Point2D):Point2D
-distMat
0..1
<<Java Class>>
DistanceMatrix
pathFinder
dist: double[][]
points: Point2D[]
N: int
size: int
DistanceMatrix(int)
addPoint(Point2D):void
getDistance(int,int):double
size():int
toString():String
<<Java Class>>
GrahamScan
pathFinder
GrahamScan()
findConvexHull(Polygon2D):Polygon2D
lowestPoint(Point2D[]):Point2D
sortByAngleFrom(Point2D[],Point2D):Point2D[]
draw(Point2D[]):void
<<Java Class>>
Polygon2D
pathFinder
Polygon2D()
Polygon2D(Point2D[])
Polygon2D(Polygon2D)
addPoint(Point2D):void
size():int
asPointsArray():Point2D[]
draw():void
drawFilled():void
toString():String
main(String[]):void
-vertices
0..*
<<Java Class>>
Point2D
pathFinder
x: double
y: double
Point2D(double,double)
Point2D(Point2D)
getX():double
getY():double
turningDirection(Point2D,Point2D,Point2D):double
distance(Point2D):double
toString():String
draw():void

and each polygon has the following relationship with the points:




## Implementation of *'GrahamScan.java'*

The graham scan algorithm was implemented in a number of steps:

1. Find the lowest leftmost point in the convex hull
2. Sort the points according to polar angle each point makes with the lowest point
3. Construct the convex stack using the turning direction formula (cross product)

The lowest leftmost point was simply computed by finding the point with the lowest $y$ value in the array. After this the point with the lowest $x$ value among the points which has the lowest $y$ value was computed and the resultant point is the lowest leftmost point in the array.

To sort the points according to polar angle, which means to sort the points clockwise or anti-clockwise taking the lowest point as the pivot point. In this case, the points have to be sorted clockwise taking the lowest leftmost point as the pivot point. The sorting was done using the BubbleSort sorting technique and the cross product to determine the turning direction of the next point in the polygon. The concept of cross product was used because calculating the angle literally using the *'Math'* class *'atan'* function is computationally expensive and an algorithm is only as useful as its complexity. The concept of cross product is pretty simple: only the sign of the result is used to determine the turning direction, the number that comes after the sign is just the area of the parallelogram that the vectors

produce and it is not really useful in this algorithm. There are only there possible conditions that needs to be considered and those are:

1. Is the sign positive → The point is turning left
2. Is the sign negative → The point is turning right
3. Is the result 0 → The point is collinear

Since the sorting is done clockwise, if the next point is turning right then there is no need to do a swap in the array of Point2D points. However if the next point is turning left then this point do need to be swapped. This process is done multiple times until there are no more swaps left to do. The resulting array is sorted according to the polar angle. Prior to actually sorting the points according to the polar angle, the lowest point has to be removed from the array so that the sorting algorithm doesn't confuse itself.

The construction of convex points involves the use of the cross product again and a stack. The stack is useful because there are two major methods called *'push'* and *'pop'* that can come in handy when checking if a point is on the convex hull or not. Basically how it works is that the first two sorted points are pushed into the stack because we can be sure that the first two points will be on the convex hull of a polygon. The next step is to use the cross product to find out if the next point is turning left or right and if one of those conditions are met, do something. In this case, if the turning direction is right push that point into the stack and if the turning direction is left, pop the last point that was pushed in and back track to the point when it was turning right. This is done in a loop until there are no more points left to check.

After the convex points have been determined, the next step is to get rid of the collinear points in the hull and this can be easily done by using the cross product. Again iterating through the hull points, if the cross product value is 0 then the middle has to be popped out of the hull stack. The resultant stack or array contains no collinear points and is capable of constructing a proper valid convex hull.

The *'sortByAngleFrom'* method in the *'GrahamScan'* class was used to sort the points by polar angle. The turning direction was computed in the *'turningDirection'* method in the *'Point2D'* class. There was only one pre-processing step involved to reduce the complecity of the problem and that was to remove the lowest point from the array before sorting them.

# *Algorithm Analysis*

The algorithm can be analysed in two parts: one that deals with the complexity of grahamscan and the other deals with the complexity of path finding algorithm.

***Grahamscan:***

The grahamscan algorithm has a complexity of $n^2$. This is because, the sorting technique used in this algorithm is BubbleSort and BubbleSort has a complexity of $n^2$ due to its nested *'for'* loop. After sorting, the program goes into hull check and the complexity of this is also $n^2$ but since they are separate, the overall complexity is of $n^2$ for the grahamscan algorithm.

***FindPath:***

The path finding algorithm used has a complexity of $n^2$ as well, theoretically. However, the complexity can even be smaller than that depending on the number of polygons (building) given in the map. This algorithm jumps from polygon to polygon until the destination point has been reached. Therefore the main *'for'* loop only needs to run 'number of polygons' times while the loop inside runs through all the vertices of a chosen polygon. the complexity is a little less than $n^2$ but maximum complexity would be of $n^2$. The two n's in $n^2$ represent a slightly different things. The first n is the number of polygons and the second n is the number of points a chosen polygon contains. To be precise the computational complexity of this algorithm is:

$$(number\ of\ polygons) * (number\ of\ vertices\ of\ a\ chosen\ polygon)$$

although the complexity is pretty small and efficient, the algorithm is not absolutely robust. There are just a few situations where this algorithm will not return a valid solution like discussed above in the Description section of this report.

However, that can be fixed by putting in a check to test if a line connecting two points of the path is intersecting a hull or not. This can be done using the cross product.

# Code Appendix

*GrahamScan.java*

```
// GrahamScan.java
// S.Somaskandan, DCU, 2016

package pathFinder;

import java.util.ArrayList;
import java.util.Stack;


import edu.princeton.cs.introcs.StdDraw;

public class GrahamScan {
	public static Polygon2D findConvexHull (Polygon2D polygon){
		Point2D[] poly_points=null;
		Point2D p0=null;
		Stack<Point2D> H=new Stack<Point2D>();

		poly_points=polygon.asPointsArray();
		p0=lowestPoint(poly_points);
		poly_points=sortByAngleFrom(poly_points, p0);

		H.push(poly_points[0]);
		H.push(poly_points[1]);

		for(int i=2;i<poly_points.length;i++){
			Point2D firstlast=H.pop();
			Point2D secondlast=H.peek();
			H.push(firstlast);
			while(Point2D.turningDirection(secondlast, firstlast,
```

```
poly_points[i]) > 0.0){
                                H.pop();
                                firstlast=H.pop();
                                secondlast=H.peek();
                                H.push(firstlast);
                        }
                        H.push(poly_points[i]);
                }

                Stack<Point2D> tmp=new Stack<Point2D>();
                while(!H.isEmpty()){
                        tmp.push(H.pop());
                }

                ArrayList<Point2D> Hull = new ArrayList<Point2D>();
                while(!tmp.isEmpty()){
                        Hull.add(tmp.pop());
                }

                for(int i=0;i<Hull.size()-2;i++){

                        if(Point2D.turningDirection(Hull.get(i), Hull.get(i+1),
Hull.get(i+2)) == 0 ){
                                Hull.remove(i+1);
                        }
                        if(Point2D.turningDirection(Hull.get(Hull.size()-
2),Hull.get(Hull.size()-1) , Hull.get(0)) == 0){
                                Hull.remove(Hull.size()-1);
                        }
                }


                Polygon2D convex=new Polygon2D(Hull.toArray(new Point2D[]{}));
                return convex;
        }


        //      Returns the lowest leftmost point in the convex hull
        private static Point2D lowestPoint (Point2D[] points){
                Point2D min=new Point2D(0,0);
                Point2D miny = points[0];
                double minx = Double.POSITIVE_INFINITY;

                int i=0;

                //      Gets the lowest y point in a given polygon
                while(i<points.length){
                        Point2D tmp= points[i];
                        if(tmp.getY() <= miny.getY()){      miny=tmp;      }
                        i++;
                }

                //      Gets the lowest x point given the lowest y value
                for(int j=0;j<points.length;j++){
                        if(points[j].getY() == miny.getY()){
                                if(points[j].getX() < minx){
                                        min=points[j];
                                        minx=points[j].getX();
                                }
                        }
```

```
                        if(points[j].getX() == min.getX()){ }
                }

                return min;
        }

        //      Sorts the given points according to the polar angle
        private static Point2D[] sortByAngleFrom (Point2D[] points, Point2D p){
                double maxy=0.0;
                int maxy_index=0;
                ArrayList<Point2D> polygon_points = new ArrayList<Point2D>();
                Point2D[] sorted=null;

                //      Removing the lowest point from the list of given points
                for(int a=0;a<points.length;a++){
                        if(points[a].getX() == p.getX() && points[a].getY() ==
p.getY()){}
                        else{ polygon_points.add(points[a]); }
                }
                points=polygon_points.toArray(new Point2D[]{});

                //      Using the cross product and the BubbleSort sorting technique, the
points are sorted clockwise
                Point2D tmp;
                for(int i=1;i<points.length;i++){
                        boolean no_swap = true;
                        for(int j=0;j<points.length-i;j++){
                                if(Point2D.turningDirection(p, points[j], points[j+1]) >
0){ // do swap
                                        tmp=points[j];
                                        points[j]=points[j+1];
                                        points[j+1]=tmp;
                                        no_swap = false;
                                }
                        }
                        if(no_swap) break;
                }

                //      Putting the lowest point back into the array
                polygon_points.clear();
                polygon_points.add(p);
                for(int a=0;a<points.length;a++){
                        polygon_points.add(points[a]);
                }

                //      Getting the max y coordinate index given the lowest x coordinate
to get rid of the middle points in the vertical axis setting the origin at the lowest
point
                for(int i=0;i<polygon_points.size();i++){
                        if(polygon_points.get(i).getX() == p.getX() &&
polygon_points.get(i).getY() > maxy){
                                maxy=polygon_points.get(i).getY();
                                maxy_index=i;
                        }
                }

                //      Removing the middle points on the vertical axis above the lowest
point
                for(int i=0;i<polygon_points.size();i++){
                        if(polygon_points.get(i).getX() == p.getX() &&
```

```
polygon_points.get(i).getY() != polygon_points.get(maxy_index).getY() &&
polygon_points.get(i).getY() != p.getY()){
                                polygon_points.remove(i);
                        }
                }

                //      Converting the ArrayList of sorted points to an array of
Point2D[]
                sorted=polygon_points.toArray(new Point2D[]{});
                return sorted;
        }

        //      Draws the given points
        public static void draw(Point2D[] points){
                StdDraw.setPenRadius(0.015);
                for(int k=0;k<points.length;k++){
                        points[k].draw();
                }
        }
}
```

*Point2D.java*

```
// Point2D.java
// S.Somaskandan, DCU, 2016

package pathFinder;

import edu.princeton.cs.introcs.StdDraw;

public class Point2D{

        private final double x;
        private final double y;

        public Point2D(double x, double y) { // constructor
                this.x = x;
                this.y = y;
        }

        public Point2D(Point2D p) { // copy constructor
                if (p == null)  System.out.println("Point2D(): null point!");
                x = p.getX();
                y = p.getY();
        }

        public double getX() { return x; }

        public double getY() { return y; }

        //      Returns the cross product value of 3 points
        public static double turningDirection(Point2D p1, Point2D p2, Point2D p3) {
                double z=((p2.getX() - p1.getX()) * (p3.getY() - p1.getY()) - (p3.getX()
- p1.getX()) * (p2.getY() - p1.getY()));
                return z;
        }

        //      Returns the distance
        public double distance(Point2D p){  return Math.sqrt((p.x-x)*(p.x-x) + (p.y-
```

```
y)*(p.y-y)); }

	@Override
	public String toString() {
		return ("(" + x + "," + y + ")");
	}

	public void draw() {
		StdDraw.point(x, y);
	}
}
```

*FindPath.java*

```
// FindPath.java
// S.Somaskandan, DCU, 2016

package pathFinder;

import java.awt.Color;
import java.util.ArrayList;

import edu.princeton.cs.introcs.StdDraw;

public class FindPath {
	private ArrayList<Point2D> Path=new ArrayList<Point2D>();
	private ArrayList<Point2D[]> Map=new ArrayList<Point2D[]>();
	private ArrayList<Point2D> Points=new ArrayList<Point2D>();
	private Point2D source;
	private Point2D destination;
	private int index;
	private static double TotalDistance=0.0;

	//	Method to load in all the points into the class to aid in the path finding
	public void insert(Polygon2D poly){
		Point2D[] temp=poly.asPointsArray();
		for(int i=0;i<temp.length;i++){
			Points.add(temp[i]);		//	Storing each point in a list
		}
		Map.add(temp);				//	Storing each polygon in
a list in the form of an array of Point2D
	}


	//	Method to add in the source point
	public void setSourcepoint(Point2D src){
		Points.add(src);
		source=src;
	}

	//	Method to add in the destination point
	public void setDestinationpoint(Point2D dst){
		Points.add(dst);
		destination=dst;
	}

	//	Method to determine which polygon contains the current point
	private int whichPolygon(Point2D p){
		Point2D[] tmp;
```

```
		for(int i=0;i<Map.size();i++){
			tmp=Map.get(i);
			for(int j=0;j<tmp.length;j++){
				if((p.getX() == tmp[j].getX()) && (p.getY() ==
tmp[j].getY())){
					index=j;
					return i; }
			}
		}
		return -1;
	}

	//	Method to find the path
	public Point2D[] findRoute(String draw){
		Point2D[] realPath;
		Point2D current;
		int nextPolygon;
		ArrayList<Integer> adjvertices=new ArrayList<Integer>();
		double[] distance;
		double previousDistance;
		double shorDist=Double.POSITIVE_INFINITY;
		int nextPoint=0;

		TSP_NN nn=new TSP_NN(this.Points);


	////////////////////////////////////////////////////////////////////////////
////////

		Path.add(source);		//	Add the source point to the list of
Path
		current=source;
		current=nn.findNextpoint(current);	//	Finds the next closest
point to the current point
		Path.add(current);

		//		Main for loop that gets the path
		for(int z=0;z<Points.size();z++){
			//	Checks if -v is present in the command line
			if(draw != null){
				StdDraw.setPenColor(Color.BLACK);
				draw_point(current);
			}

			nextPolygon=this.whichPolygon(current);			//
	Calls the whichPolygon method to find out the polygon on which the current lies
on.
			previousDistance=current.distance(this.destination);	//
	Stores the distance of the current point to the destination point

			//	Sets up the graph for a polygon connecting the adjacent
vertices
			Graph g=new Graph(Map.get(nextPolygon).length);
			for(int j=0;j<Map.get(nextPolygon).length-1;j++){ g.addEdge(j,
j+1); }
			g.addEdge(Map.get(nextPolygon).length-1, 0);
			adjvertices=g.adj(index);		//	Gets the adjacent
vertices to the current vertex
```

```
                        //      Checks if -v is present in the command line
                        if(draw != null){
                                for(int i=0;i<adjvertices.size();i++){
                                        StdDraw.setPenColor(Color.CYAN);
                                        draw_point(Map.get(nextPolygon)[adjvertices.get(i)]);
                                }
                                draw(Path.toArray(new Point2D[]{}));
                        }

                        //      Initializing the distance array with the size of the
adjacent vertices list
                        distance=new double[adjvertices.size()];

                        //      Getting the distance from the adjacent vertices to the
destination point and storing them in the distance array
                        for(int i=0;i<adjvertices.size();i++){
                                Point2D adjacent=new
Point2D(Map.get(nextPolygon)[adjvertices.get(i)]);
                                distance[i]=adjacent.distance(this.destination);
                                nn.setTrue(Map.get(nextPolygon)[adjvertices.get(i)]);
                        }

                        //      Getting the shortest distance from the 'distance' array
                        for(int j=0;j<distance.length;j++){
                                if(distance[j] < shorDist){
                                        shorDist=distance[j];
                                        nextPoint=adjvertices.get(j);
                                }
                        }

                        //      If the new distance is less than the current distance,
                        //      add the calculated next point to the array of path after
some more checks
                        if(previousDistance < shorDist){
                                //      Setting the whole polygon to true
                                for(int i=0;i<Map.get(nextPolygon).length;i++){
                                        nn.setTrue(Map.get(nextPolygon)[i]);
                                }
                                Point2D next=nn.findNextpoint(current);          //
        Finds the next closest point to the current point

                                //      If the distance from the next point to the
destination is less than previous distance
                                //      let the next point equal to the current point so it
can be added to the list of Path
                                //      Else find a different next point
                                if(next.distance(this.destination) < previousDistance){
                                        current=next;
                                }
                                else{
                                        nn.setTrue(next);
                                        next=nn.findNextpoint(current);
                                }
                                Path.add(current);                                      //
        Add the current point
                                if(current==destination){break;}          //      if the
current point is equal to the destination point, then break;
                        }
                        else{
                                Path.add(Map.get(nextPolygon)[nextPoint]);
```

```
                        current=Map.get(nextPolygon)[nextPoint];
                        if(current==destination){break;}
                    }
                    shorDist=Double.POSITIVE_INFINITY;              //    Reset
shorDist so that it can be used for next calculation
            }

            realPath=Path.toArray(new Point2D[]{});                 //    Converting
the ArrayList 'Path' to an array of Pointr2D[]
            //    Checks if -v is present in the command line
            if(draw != null){
                    StdDraw.setPenColor(Color.CYAN);
                    draw(new Point2D[]{realPath[realPath.length-2],
realPath[realPath.length-1]});
            }
            //    Calculates the distance
            setDistance(realPath);
            return realPath;                                                    //
      Returns the array of Point2D[]


      ////////////////////////////////////////////////////////////////////////////
////////
      }

      //    Calculates the total distance of the path and stores the value inside a
global variable
      private static void setDistance(Point2D[] points){
            for(int i=0;i<points.length-1;i++){
                    TotalDistance=TotalDistance+points[i].distance(points[i+1]);
            }
      }

      //    Return the total distance calculated
      public double getDistance(){
            return TotalDistance;
      }

      //    Draws the path without connecting the source to the destination
      public void draw(Point2D[] vertices) {
            for (int i=0; i<vertices.length-1; i++) {
                    StdDraw.pause(300);
                    StdDraw.setPenRadius(0.009);
                    StdDraw.line(vertices[i].getX(), vertices[i].getY(),
vertices[i+1].getX(), vertices[i+1].getY());
            }
      }

      //    Draws a point
      public void draw_point(Point2D point) {
            StdDraw.pause(300);
            StdDraw.setPenRadius(0.02);
            StdDraw.point(point.getX(), point.getY());
      }


}
```

*Graph.java*

```
// Graph.java
// S.Somaskandan, DCU, 2016

package pathFinder;

import java.util.ArrayList;

public class Graph {
	private final int v;
	private int E;
	private boolean[][] adj;

	//	Constructor
	public Graph(int v){
		this.v=v;			// Number of vertices
		this.E=0;			// Initializing the number of edges
		adj=new boolean[v][v];		// Initializing the array of adjacent
vertices
	}

	//	Returns the number of vertices
	public int v(){ return v; }

	//	Returns the number of edges
	public int E(){ return E; }

	//	Adding an edge using two vertices
	public void addEdge(int v, int w){
		adj[v][w]=true;			//	True if edge is present
		E++;				//	Edge is incremented by 1
	}

	//	Return a list of adjacent vertices to the current vertex
	public ArrayList<Integer> adj(int v){
		ArrayList<Integer> adjvertices=new ArrayList<Integer>();
		if(v>=this.v){ return null; }
		for(int i=0;i<this.v;i++){
			if(adj[v][i] || adj[i][v]){
				adjvertices.add(i);
			}
		}

		return adjvertices;
	}
}
```

*TSP_NN.java*

```
// TSP_NN.java
// S.Somaskandan, DCU, 2016

package pathFinder;

import java.util.ArrayList;

public class TSP_NN {
	private ArrayList<Point2D> conPoints=new ArrayList<Point2D>();
	private DistanceMatrix distMat;
	private boolean[] check;
```

```
	//	Constructor to load in the points
	public TSP_NN( ArrayList<Point2D> p){
		this.conPoints=p;
		this.distMat= new DistanceMatrix(this.conPoints.size());
		calculateDistance(this.distMat);
		check=new boolean[p.size()];
	}

	//	Initializing the DistanceMatrix class with the points
	private void calculateDistance(DistanceMatrix dist){
		Point2D[] tmp=this.conPoints.toArray(new Point2D[]{});
		for(int i=0;i<this.conPoints.size();i++){
			dist.addPoint(tmp[i]);
		}
	}

	//	Sets the given point true so that it won't be visited again
	public void setTrue(Point2D t){
		this.check[conPoints.indexOf(t)]=true;
	}

	//	Computes the next point given the current point
	public Point2D findNextpoint(Point2D current){
		double shortestDistance=Double.POSITIVE_INFINITY;
		double distance=0.0;
		int nextPoint=0;

		this.check[conPoints.indexOf(current)]=true;
		for(int i=0;i<this.conPoints.size();i++){
			if(!this.check[i]){                                    //
	Checks if a point has already been visited

	distance=this.distMat.getDistance(conPoints.indexOf(current), i);

                        //      Computes the next point considering the distance
between the current point and a different unvisited point in the map
                        if(distance < shortestDistance){
                                shortestDistance=distance;
                                nextPoint=i;
                        }
                }
        }
        return this.conPoints.get(nextPoint);
    }
}
```

*DistanceMatrix.java*

```
// DistanceMatrix.java
// S.Somaskandan, DCU, 2016

package pathFinder;

public class DistanceMatrix {

	private double[][] dist;    // distances between points
	private Point2D[] points;   // set of points (city locations)
	private int N;                              // current number of points
```

```
        private int size;                   // total number of points

        public DistanceMatrix(int size) { // constructor
                dist = new double[size][size];
                points = new Point2D[size];
                this.size = size;
        }

        public void addPoint(Point2D p) {
                if (N == size) return;

                points[N] = new Point2D(p); // make a copy
                // calculate distances from new point to points previously added
                if (N > 0) {
                        for (int i=0; i<N; i++) {
                                dist[N][i] = p.distance(points[i]);
                                dist[i][N] = p.distance(points[i]);
                        }
                }
                N++;
        }

        //      Returns the distance between 2 given points in the map
        public double getDistance(int i, int j) {
                return dist[i][j];
        }

        //      Returns the total number of points in the map
        public int size() {
                return size;
        }

        @Override
        public String toString() {
                String s = "";
                for (int i=0; i<size; i++) {
                        for (int j=0; j<size; j++)
                                s += String.format( "%.3f", dist[i][j]) + " ";
                        s += "\n";
                }
                return s;
        }

}
```